Design of the SRM requests and request scheduler.

Introduction

SRM Request Scheduler provides a mechanism for execution of SRM requests in a manner that yields maximum utilization of the storage system resources without overloading the system. Main resources managed by the scheduler are a pool of execution threads and queues of requests. It attempts to provide fairness of the execution of the requests between multiple users. If one user submits a request with a ten thousand Execution Requests in it and the next user submits a request with 1 Execution Request, the second request does not wait for the completion of all ten thousand Execution Requests before being executed. Request Scheduler paces the transition of the requests into Ready state (which means delivery of the transfer URLs to the users), to control the execution of the users initiated transfers.

SRM provides a persistent storage of the SRM requests states, so that the system should be able to survive reboots.

This scheduler is non-preemptive, meaning that if a job is being executed, it can not be temporarily stopped to give way to the execution of the other job.

Description

Job

Job is an abstract class that has a number of properties and run method. Job has a state property. The goal of the scheduler it so execute job's run method (potentially more than once) and perform other manipulations until job reached one of its final states (value of the state property) . Once job is scheduled to be executed by the scheduler the scheduler places Job in a queue, and eventually executes job's run method, using one of the

Scheduler's pooled thread. Depending on the state of the job upon completion of the execution, job can be determined to have been completely executed, partially executed, failed, canceled, and so on. Jobs in a final state, that is in Done, Failed or Canceled state, are not managed by the Scheduler. More detailed are bellow.

Each SRM request or file request executed by the dCache SRM is a subclass of Job.

Each job has the following properties:

Job Id is a long integer uniquely identifying each instance of the Job

Job Submitter (Job owner, creator, user) is a unique string uniquely identifying the submitter / creator of the job. All the jobs created by the same submitter should return the same submitter string, so that the scheduler could use this info in counting the number of the jobs running corresponding to each job submitter and could make a fair scheduling decisions, on basis of some measure of fairness.

Job Priority is a non-negative integer, used in making the scheduling decision. The higher the job priority the more likely it will be executed sooner compared with the Jobs scheduled to be executed at the same time.

Job Creation time (in milliseconds as reported by System.currentimemillis()) and **lifetime** (in milliseconds) used to determine when to change the state of the job still not in final state to Canceled state.

Job's Number of retries – number of times the the job execution failed in non-fatal manner and was scheduled to be retied again.
Job's Max number of retries maximum number of times to attempt to execute a job after the run method throws non fatal exceptions, before failing a job.
Next Job Id was planned to be used for the creation of the linked lists of the jobs, by the queues.

Job States

Job State is represented by org.dcache.srm.scheduler.State class.
The Job has (or is in) a state. The states are

1. Pending – Initial state of the Job,
2. TQueued - Waiting on a Thread Queue for a Thread in the Thread Pool ,
3. PriorityTQueued – Waiting on a Priority Thread Queue for a Thread in the Thread Pool.
3. Running – Utilizing A Thread from the Thread Pool,
4. RunningWithoutThread, a job is not running withing the thread, but job is being processed by something outside the scheduler, by still should be counted towards the total number of running jobs, thus the total number of Running and RunningWithoutThread jobs should be less than MaximumNumberOfRunning Jobs
5. AsyncWait -Waiting For Async Notification, upon the notification, the job is placed on the the PriorityTQueue
6. ReadyQueued - Waiting on Ready Queue till Job may become Ready,
7. Ready , all transfer URLs are available, the server is ready to perform transfer.
8. Transferring – client indicates that the Job file is being transfered, or server is performing transfer itself.
9. RetryWait – nonfatal failure of Job execution , timer is started, upon expiration of which the Job is added to the TPriorityQueue.
10. Failed – fatal failure in execution, Final State,
11. Done – Job has either completed or Ready Job has been set to Done by the user, Final State.
12. Canceled – Job has been canceled by the user, Final State.

Scheduler

Scheduler manages execution of Jobs. Scheduler internally utilizes queues, timers and thread pools to provide a controlled Job execution environment.

Major Components of the Scheduler

1. **Queues**

   by the queue we understand an ordered collection of jobs. When the job is extracted from the queue, it is not necessarily the first job in the collection is extracted, but all jobs can be considered and the decision on the extraction is made on basis of many factors, including the position of the job in the queue.

Scheduler utilizes three Queues called “Thread Queue”, “Priority Thread Queue” and “Ready Queue”.

2. **Threads and a Thread pool.**
   A number of threads, collectively called a pool of threads, is used for the execution of the jobs, Once the execution is completed, the thread is returned to the pools for further utilization. The total number of threads is not allowed to grow beyond the fixed limit.
3. **Timers**
   A timer is an object that, once created, executes a certain code at a specified moment in time called timer expiration time.

Execution of the Job by the Scheduler

Execution of Job requires utilization of certain storage (disk,tape etc), memory, processor and network resources. One of the requirements is that upon the completion of Job execution (successful or not) all resources should be released. The exception is the storage resources occupied by a file created in a permanent storage as a result of the request. The goal of the execution is to bring the job from the initial state of the job, “Pending”, to one of the final states “Canceled”, “Failed” or “Done”.

Job is initially in the Pending state and is is scheduled with a particular scheduler by the execution of the scheduler's *schedule* method. After that the job is placed in the Thread Queue, and its state is set to TQueued. Thread Queue itself has a limit on number of elements it may hold. If the limit is already reached, Job is failed.

As long as there are threads available in the thread pool and the jobs are present in the Priority Thread Queue or a Thread queue, the scheduler, weights the queued jobs using plugable JobPriorityPolicyInterface, which evaluates the immediate priority for each of the job in the PriorityThreadQueue or, if it is empty, for each of the Job in the Thread queue. The following parameters are considered by the JobPriorityPolicyInterface

1. Queue length
2. Job Position in the queue,
3. number of jobs running submitter by the same user
4. Maximum number of Jobs allowed by the same user

5. Job itself, so that its properties (priority) can be examined.

The job with the highest weight is removed from the queue and executes the scheduler runs the run method of the job with the highest value using one of the threads of the pool.

After the completion of the run method Job can be in a number of states.

If it is in the Done, Failed or Canceled state, no more processing is required.

If it is still in a Running state, the Job is placed in the Ready queue and its state is set to RQueued.

If it is in AsyncWait state, the scheduler needs to wait until and internal even will reschedule the job by running the scheduler method again. In this case a job is placed in the PriorityThreadQueue and jobs state is set to PriorityTQueued. Than it will eventually get its run method run again.

If the run method execution completes with an exception, and exception is a NonFatalException, the job number of retries property is increased and if it less than the maximum number of retries, it's put in the RetryWait state, and a Retry timer is started. If the number of retries is greater than the maximum number retries (see scheduler properties below), the job is failed. Upon the expiration of the retry timeout, the timer schedules the Job, running the schedule method of the scheduler, and it is places in the priority thread queue.

The Scheduler allows only the limited number of jobs in the Ready state. If the number of jobs in the ready state is less than the maximum number, the Jobs in the ready queue weighted using the same process as described for the Jobs in the Thread queue, and the job with the highest value is set to Ready state. Job state to Transferring and then to Done, or user can change the state to Done directly.. Changing Job state to Running does not reduce the total number of Ready jobs from the point of view of scheduler.

The Job in the Ready state could be set to Done state by external event.

The Ready State is used to model the state of the SRM Put and Get file requests when the TURL is given out to the client and it is client's responsibility to perform the transfer. Limiting the total number of the jobs in the ready state we limit the maximum number of simultaneous transfers performed by the client thus preventing the overall

If the job lifetime expires (creation time + lifetime > current time) the Job is either set to Done if it was in Ready state or to Failed state otherwise.

Every Job should move to the Canceled, Done or Failed state by the end of the execution or of its lifetime.

Job State Transitions

Another way to think of the execution of the Jobs by the scheduler is in terms of the state transitions. Here are most of the common state transitions

Any State -> Canceled: User canceled request

State -> Failed: Job lifetime has expired. This transition will happen for Pending, TQueued, Running, AsyncWait,RQueued and RetryWait states.

(Ready/Transferring) State -> Done: Job lifetime has expired. This transition will happen for Ready and Transferring states.

(Ready/Transferring) State -> Done: Job has been set to Done by the user . This transition will happen for Ready and Transferring states.

Pending->Running: Number of Running Job is less then number of threads in the pool.
Pending ->TQueued: No threads are left in the thread pool. Job is passed to the Scheduler. Scheduler has placed Job on the Thread Queue.
Pending -> Failed: Job is passed to the Scheduler. Thread Queue size limit is reached.

(PriorityTQueued/TQueued->Running): Job is assigned a thread from the Thread Pool for utilization.

Running->AsyncWait: Job Execution Code Starts a Wait for Asynchronous Notification
Running->RetryWait: Job Execution code fails with nonfatal error and the number of performed retries is less then max allowed. The timer with retry timeout is started
Running->Failed: Job Execution Code fails with fatal error or code fails with nonfatal error and max retries limit is reached
Running->RQueued: Job Execution Code completed successfully,Number of Ready or Transferring request is equal to the max number of ready requests.
Running->Ready: Job Execution Code completed successfully,Number of Ready or

Transferring request is less then the max number of ready requests.
Running->Ready: Job can be put into the Done state by the Job Execution Code itself.

AsyncWait->PriorityTQueued: Job received asynchronous notification from external source

RetryWait->PriorityTQueued: retry timeout expired

RQueued->Ready: Number of Ready or Transferring request is or has become less then the max number of ready requests.

Ready->Transferring: User set status of Ready Job to Transferring.

Ready->Done:User set status of Ready Job to Done or lifetime of the job expires.

Transferring->Done:User set status of Ready Job to Done.

Figure 1 bellow illustrates some of the State Transitions of Jobs.

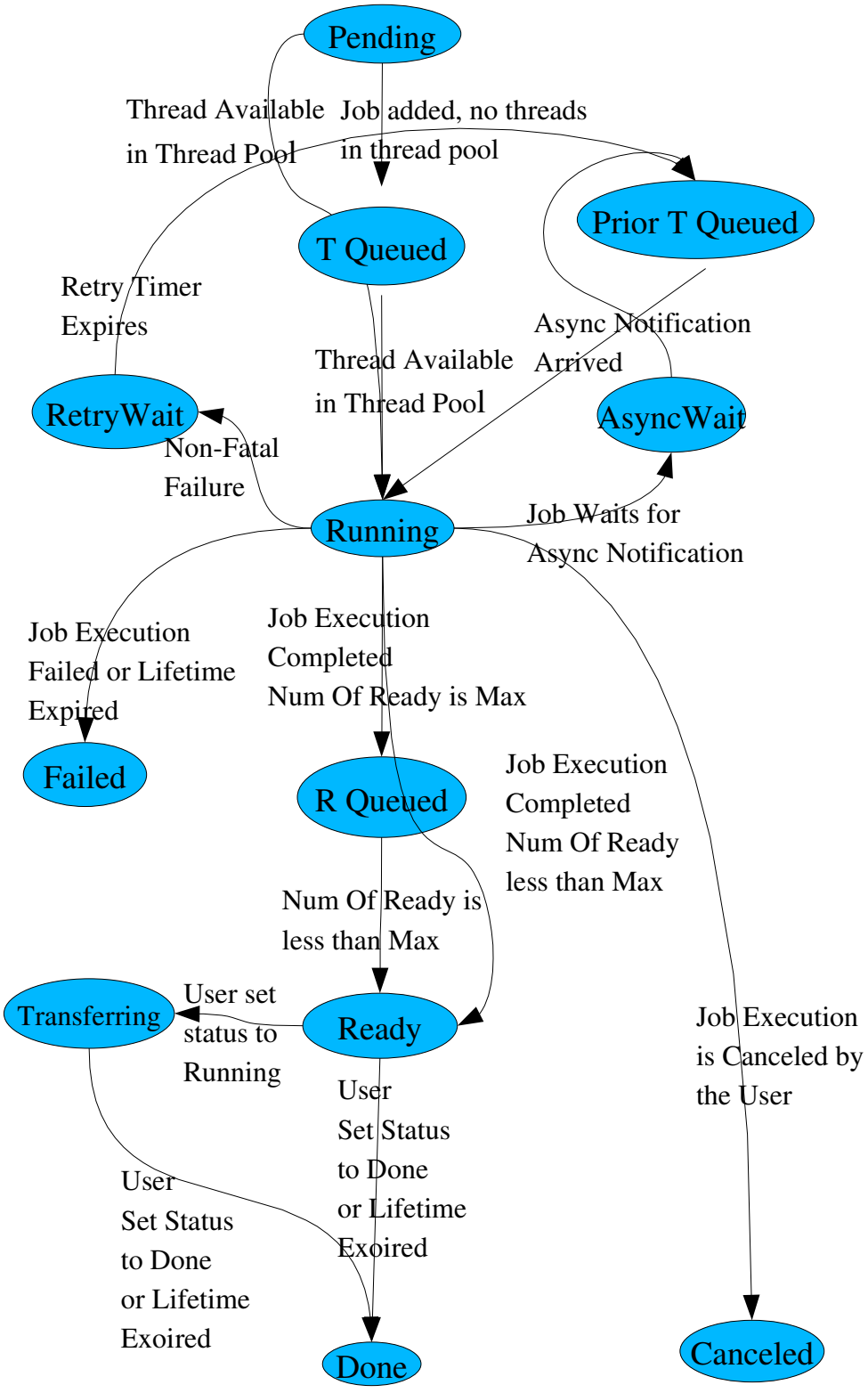
Pending
Thread Available
in Thread Pool
Job added, no threads
in thread pool
T Queued
Prior T Queued
Retry Timer
Expires
Async Notification
Arrived
Thread Available
in Thread Pool
RetryWait
AsyncWait
Non-Fatal
Failure
Running
Job Waits for
Async Notification
Job Execution
Failed or Lifetime
Expired
Job Execution
Completed
Num Of Ready is Max
Failed
R Queued
Job Execution
Completed
Num Of Ready
less than Max
Num Of Ready is
less than Max
Transferring
User set
status to
Running
Ready
Job Execution
is Canceled by
the User
User
Set Status
to Done
or Lifetime
Exoired
User
Set Status
to Done
or Lifetime
Exoired
Done
Canceled

Fig1. States

Scheduler properties

Scheduler ID a String uniquely identifying the Scheduler within this instance of JVM and within dCache. This ID should not change after restart, as it is used to determine the jobs that "belong" to this scheduler and should continue to be executed.

The following important parameters can be configured.
MaxThreadQueueSize, Thread Queue size
ThreadPoolSize, Thread Pool size,
MaxAsyncWaitJobs,
MaxReadyQueueSize, Ready Queue Size,
MaxReadyJobs, Max Number of Ready requests,
RetryTimeout, Retry Timeout in milliseconds,
MaxRetryWaitJobs, Maximum number of jobs in the retry wait state
MaxNumberOfRetries, Maximum Number or Retries
MaxRunningByOwner, MaxRunningWithoutThreadByOwner

For different types of the request separate instances of the SRM Scheduler are created so that the exectuion parameters for each type of the request can be fine tuned separately. So the instance of the SRM server is running Get, Put, Copy, BringOnline, ReserveSpace and Ls request schedulers. The following parameters related to the Get request scheduler are configurable via dCacheSetup, substituting the "Put", "Copy" and "BringOnline" for "Get" will produce the names for parameters controlling behavior of the other requests.

srmGetLifeTime
srmGetReqThreadQueueSize
srmGetReqThreadPoolSize
srmGetReqMaxWaitingRequests
srmGetReqReadyQueueSize
srmGetReqMaxReadyRequests
srmGetReqMaxNumberOfRetries
srmGetReqRetryTimeout
srmGetReqMaxNumOfRunningBySameOwner

Job Storage

A persistent job storage for the job. Since the actual job implementation is not known to the scheduler, the implementation of the job storage(s) should be provided by the implementors of the jobs itself. Every time the job state changes in any meaningful way the job should be saved. Since job knows the about its persistent storage, the job is saved by calling its save() method. Also the usage of the persistent storage allows to reduce the usage of the volatile computer memory keeping the job id in memory only when job is not active. For example the job queue needs to keep the id of the first and last job in the memory only. More information is in dCache SRM Design Document at http://home.fnal.gov/%7Etimur/srm/review/dCacheSRMDesign.pdf .